VDMTools: advances in support for formal modeling in VDM

John Fitzgerald
School of Computing Science
Newcastle University, UK
John.Fitzgerald@ncl.ac.uk

Peter Gorm Larsen
Engineering College of Aarhus,
Denmark
pgl@iha.dk

Shin Sahara
CSK Systems, Japan
shin-sahara@kuzo.csk.com

Abstract

We describe the current status of "VDMTools", a group of tools supporting the analysis of system models expressed in the formal language of the Vienna Development Method. Three dialects of the language are supported: the ISO standard VDM specification language with support for modular structuring, the extension VDM++ which supports object-oriented structuring and concurrency, and a version extending VDM++ with features for modeling and analysing distributed embedded real-time systems. VDMTools provides extensive static semantics checking, automatic code generation, round-trip mapping to UML class diagrams, documentation support, test coverage analysis and debugging support. The tools' focus is on supporting the cost-effective development and exploitation of formal models in industrial settings. The paper presents the components of VDMTools and reports recent experience using them for the development of large models.

Keywords Formal Methods, Vienna Development Method, VDM, Validation, Tool support

1. Background

Formal methods are mathematically-based techniques for the modeling, analysis and development of software and systems (24; 14; 9; 12). Their use is motivated by the expectation that, as in other engineering disciplines, performing an appropriate mathematical analysis can contribute to the reliability and robustness of a design. Formal methods can be used at various levels of rigour and, in industrial applications, it is paramount to be able to strike a proper balance between the effort spent on the use of formal techniques and the insight gained (16).

The Vienna Development Method (VDM) is one of the most mature formal methods, primarily intended for the modeling and subsequent development of functional aspects of software systems (38). Applying VDM involves developing a system model expressed in a (formal) modeling language. The language's formality means that the full range of analytic techniques, from testing to formal mathematical proof, can be applied to validate the model, or to verify the correctness of the model with respect to an existing statement of requirements or design. Three dialects of the VDM modeling language are in use, each supporting a different form of system:

1. VDM-SL (19) provides facilities for the functional specification of sequential systems with basic support for modular structuring. It has been standardised under the auspices of the International Organization for Standardization (ISO) (39).
2. VDM++ (18) extends VDM-SL with features for object-oriented modeling and concurrency (40).
3. VICE (VDM++ In Constrained Environments) further extends VDM++ with features for describing real-time computations (36) and distributed systems (46).

VDM modelling techniques have been used in a variety of ways within software development processes. One very pragmatic approach is to develop VDM-SL system models and analyse them by prototyping in a programming language. For example Kans and Hayton (27) report on the use of ABC+ to prototype VDM-SL specifications for educational purposes; Borba and Meira (6; 7) describe a prototype semi-automatic translation system producing Lazy ML code from VDM-SL models. These approaches have the drawback that another formalism (in these examples ABC+ or Lazy ML) must be introduced to cope with the semantics of the VDM-SL specifications. Furthermore the support provided for prototyping in these tools is not tailored for VDM-SL but rather depends on the programming language in use. Similar issues have been raised in other formal specification language notations (23; 44; 10) and, more generally, there has been debate around the value of executable specifications (25; 22; 2; 20). Liu's work based on the SOFL language (33) contains elements derived from VDM and supports direct animation of models from systematically derived execution scenarios.

A more formal approach to the analysis of VDM models exploits the mathematical semantics by using formal proof to support the verification of key system properties in formal models. Many texts cover this area e.g. (5) provides a sound constructive approach to proof in VDM. Much of the mathematical foundation has been laid to support fully formal development in which the design steps from an abstract model to an implementation are formally verified (26). Even with state-of-the-art tools, carrying out formal proofs on the industrial scale is still expensive, although success stories have begun to appear, such as those surrounding B (1; 3), which suggest that the approach is becoming cost-effective.

VDMTools are a development of the IFAD VDM Toolbox (15) to support the three dialects of VDM listed above and intended to support a pragmatic approach (35) to the construction and analysis of formal models. The tools provide syntax checking, extensive static semantics checking and documentation support. Furthermore they support the validation of specifications written in a (large) executable subset of VDM-SL using testing and debugging techniques. We have applied VDMTools successfully to models up to 100k lines of VDM model in size.

2. Main Characteristics of VDM Modeling Notations

In this section we review the more distinctive features of the VDM modeling notations which pose challenges for tool support.

VDM models center on data type definitions built from a repertoire of abstract base types and type constructors including union and record types, sets, sequences and mappings. Any type definition in VDM can be augmented with an *invariant* – a Boolean formula describing a property that must be respected by all elements of the newly defined type. Consider, for example, the following type definition in VDM-SL:

```
PosReal = real
inv r == r >= 0
```

This defines a new type `PosReal` containing all real numbers satisfying the invariant property (in this case that they are larger than or equal to zero). Such a type could then, for example, be used in the definition of a function, such as the following, which defines a function `ImplSort`:

```
functions

ImplSort(l: seq of PosReal)
         r: seq of PosReal
post IsPermutation(r,l) and
     IsOrdered(r);
```

The input `l` of the function `ImplSort` is a sequence of `PosReal`, so the function body can rely on the fact that the elements of `l` respect the invariant on that type. Note also that this function definition does not contain an expression or algorithm for calculating the result `r` from the input. Instead a postcondition characterizes the result in terms of the properties that are required of the result, namely that it should be a permutation of the input and that it should be ordered. This property is stated using two Boolean auxiliary functions: `IsPermutation` and `IsOrdered`.

Functions may also be defined explicitly. For example, the `IsOrdered` auxiliary function might be defined as follows:

```
IsOrdered: seq of real -> bool
```

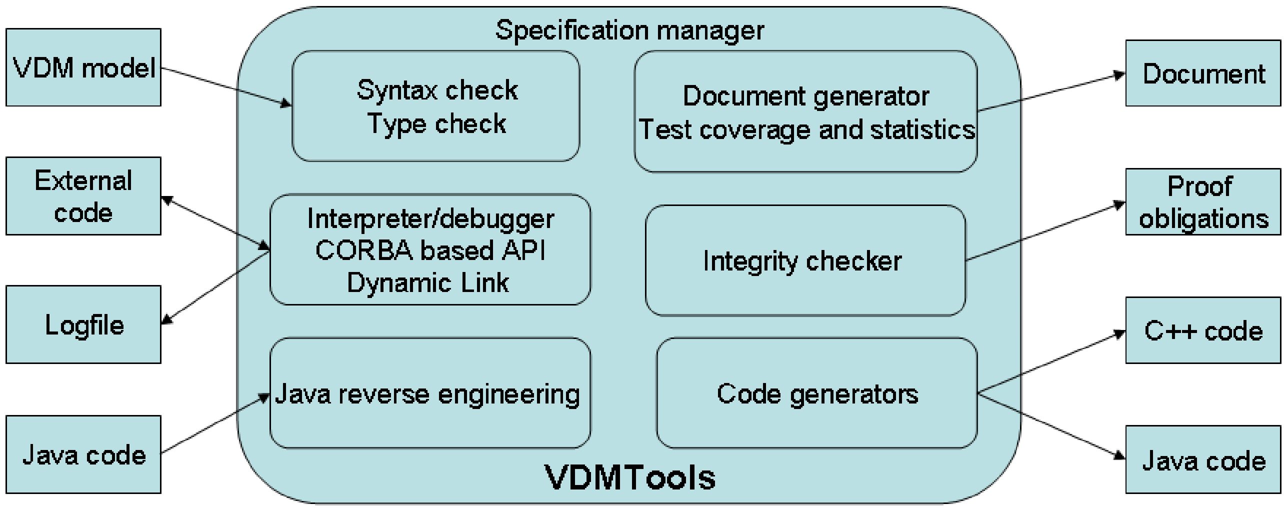

Figure 1. VDMTools Overview

```
IsOrdered(l) ==
  forall i,j in set inds l &
     i > j => l(i) >= l(j);
```

Here the body of the function is a Boolean expression stating that, for any indices in the sequence, the value at the high index is itself at least as large as the value at the lower index. As this example shows, because of the importance of data type invariants and the possibility of implicit definition, logic is a cornerstone of VDM.

In addition to the forms of construct introduced here, VDM also has a built-in notion of persistent state variables that can be modified by operations which may themselves be specified implicitly or explicitly.

The VDM++ modeling language also supports the description of object-oriented and concurrent systems, with persistent state modeled as instance variables. Logic expressions are used to describe the synchronisation constraints on threads. The VICE extension further supports the modeling of time and the deployment of functionality to resources in a distributed system architecture.

3. Working with VDMTools

Figure 1 gives an overview of the structure of VDMTools. Figure 2 is a screen dump showing part of the graphical user interface through which the major tool functions can be applied to a model. We will present the tool components by describing the way in which

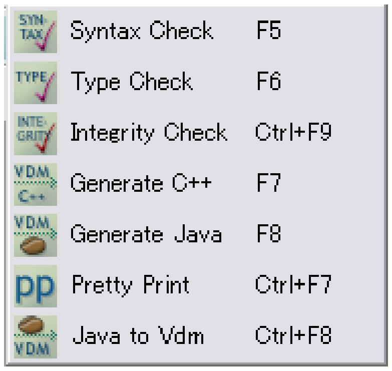

Figure 2. VDMTools Actions Menu

they are typically used during the development of VDM models. The underlying principle in the tools' development has been to support industrial use. At various points, we have emphasized robustness over the provision of advanced functionality.

VDMTools support concurrent development and allow models to be split among several files. Models can be written in an ASCII representation, Unicode (if, for example, Japanese characters are desired), or a mix of source code and comment using in a literate programming style with LaTeX macros. This allows standard version control system such as CVS to be readily applied to the model files. Alternatively VDM models can be produced directly using Microsoft Word, provided they are stored in Rich Text Format (`rtf`) and use special predefined styles.

3.1 Producing Documents

Since a formal model provides an unambiguous system description, it can serve as the basis for technical documentation of the specification and the implemented system. We see VDM models as an integral part of a system's documentation which also includes such material as informal specifications, graphical structural models and model annotations. VDMTools supports this idea, not only by allowing models to be composed from multiple source files, but also by including a pretty-printing facility whereby ASCII or Unicode models can be translated to LaTeX source that renders the model in the mathematical syntax preferred by some users. This also supports the automatic generation of LaTeX indexes for models. We have found these facilities to be essential when handling large specification documents.

3.2 Syntax Checking

VDMTools supports syntax checking of VDM models with positional error reporting supported by an indication of error points in a source window. VDMTools may be configured so that a user's favorite external editor can then be invoked with the file shown in a source window.

3.3 Type Checking

Having established a VDM model as syntactically correct, the next step is to check its static semantics (type errors, scope errors, etc.). VDM has a powerful type system supporting constructors that are challenging from a traditional type-checking perspective. These include union types and recursively defined types. VDMTools includes a static semantics analyser that is able to check for a large number of the static semantic errors well known from normal programming language type checkers. These include incorrect values applied to function calls, badly typed assignments, use of undefined variables and module imports/exports. Since the VDM modelling languages have formal static semantics, it is also possible to provide extended checks. In all these cases the types involved can be arbitrarily complex compositions of any of the standard VDM types.

For large models it is important to be able to check smaller parts of the specification in isolation before they are integrated. The static semantics analyser can check VDM-SL modules and VDM++ classes seperatedly but when checking several modules/classes will also check the consistency of the module/class interfaces.

3.4 Integrity Checking

In addition to static type checking, VDM defines checks to avoid semantic inconsistency and potential sources of run-time errors. These checks are termed *proof obligation*. For example, a proof obligation (known as the *satisfiability* obligation) requires that all functions and operations respect the data type invariants on their outputs and persistent state values. Run-time error checking includes the obligation to show that partial operators are applied safely, for example avoiding division by zero or array bounds violation. The integrity checker in VDMTools (4) detects all places where proof obligations may be violated and generates a description of the obligation. This may subsequently be checked by inspection, or taken over to an external proof tool and formally verified (47).

3.5 Executing VDM Models

To aid the understanding of complex models VDMTools supports execution and debugging. The debugger supports many of the facilities known from debuggers of programming languages such as setting breakpoints (at functions and operations), stepping (performed at expression and statement level) and inspecting the current calling stack. Furthermore the interpreter supports VDM-specific facilities such as dynamic checking of type invariants and checking pre- and post-conditions on function and operation calls.

The interpreter also supports the incorporation of external (legacy) code in C++ using *dynamic link libraries* (21). The external code is compiled into a `.ddl` and the interface of that code is described at the VDM level so that the user can access its functionality directly. In addition, standard libraries for mathematical functionality and input/output are directly built in to VDMTools so the user does not need to define these.

One of the advantages of VDMTools compared to other prototype-based approaches is the large subset of VDM that is supported for execution. All the VDM constructs supported in other prototype-based approaches (e.g., (6; 27) may be executed directly in VDMTools. Furthermore more advanced constructs are supported, including higher order functions, polymorphic functions, complex (loose) pattern matching, comprehension expressions for mappings, sets and sequences, lambda expressions and exception handling (32). VDM, being a modeling rather than a programming language, contains several constructs that are not executable, so VDMTools does not support execution of expressions in which local variables range over entire data types (which are unbounded in VDM) and purely implicitly defined functions and operations. In fact, research indicates that larger parts of the language could be supported by the interpreter (20).

3.6 External Access via an API

When models have been checked for internal consistency, it is important that they should be *validated*. By validation we mean the process of increasing confidence that a model is faithful to the stakeholders' expectations (e.g. that it embodies critical properties and does indeed describe the behaviour of the system under consideration). Stakeholders are rarely expert formal modellers, and so VDMTools includes a CORBA-based Application Programmer Interface (API) that can be accessed by either external C++ or Java, allowing the full functionality of VDMTools to be accessed by an external application such as a graphical user interface designed for domain experts unfamiliar with the modeling notations. Feedback from scenarios performed with such domain experts can be incorporated immediately and updated without the need for new compilation of the application.

3.7 Support for Validating Distributed Models

The VICE version of VDMTools has an interpreter that automatically produces an external logfile containing all the events observed during an execution of a model. Each event in this logfile is tagged by the time at which it occurs and, in the case of a distributed system model, also the resource on which the event appeared (45). This information can then be displayed graphically so potential bottlenecks in a proposed system architecture can be discovered at a very early stage in development. See Figure 3 for an example of such a graphical overview. Time is shown on the x-axis; the computing resources in the distributed system are listed on the y-axis. Thick lines indicate that the resource is busy and thin lines between resources indicate messages that are passed over a communication medium. The large arrows indicate swapping in and out of task on a specific resource.

A further extension in this direction permits the formulation of validation conjectures (system-level timing requirements) and automated checking that a system logfile respects them. Violations can then be identified graphically to the user on the trace display (17). This feature has been experimentally evaluated but is not yet integrated into VDMTools.

3.8 Systematic Testing

In practice, systematic testing is the most widely-used technique for gaining confidence in the correctness of a large formal model in VDM. The interpreter supports interactive testing of models, and has an additional test support mode. This facility allows a test suite to be set up for the specification using standard shell scripts, allowing thorough batch-mode testing of reasonably large models and the provision of test coverage statistics.

3.9 Code Generators

The VDMTools contain code generators for a large subset of the VDM modeling language (around 95%), producing C++ and Java code. Applying the code generator to a type correct VDM model yields an implementation rapidly although the generated code will most likely be slower than manually crafted code. Thus, the utility of this feature depends on the nature of the application and its performance requirements. As explained in Section 3.11 below the VDMTools features are themselves developed from VDM models; for parts of these models, the code generators have actually been used to produce the VDMTools production code itself. Note also that the code generators enable the user to update parts of the generated code manually and take that new code into account next time code is generated.

3.10 Reverse Engineering Support for Java

VDMTools incorporates a beta-version of a feature that allows code in a Java subset to be reverse engineered to low-level VDM++ models. The subset does not support the GUI libraries because those parts can better be used with the CORBA-based API instead. The current beta-version here is expected to be most useful for research and experimentation.

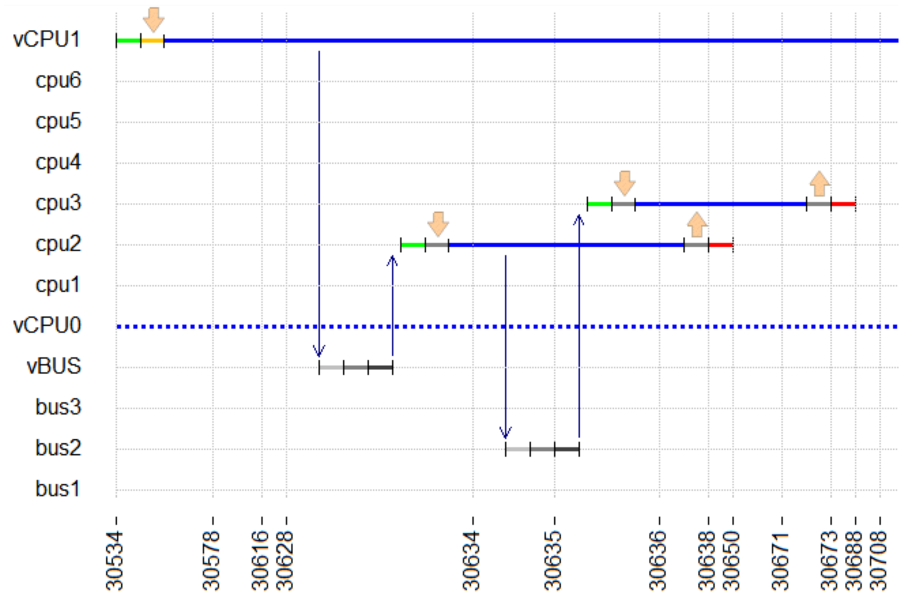

Figure 3. Graphical overview of distributed system model activity during execution

3.11 Implementation

Once a formal model has been suitably validated (using proof obligation generation, reviews and testing), implementation is the next step. In our own work we have found that implementing a formal model is an almost mechanical, but mainly manual, effort once an implementation strategy has been adopted. In the development of VDMTools themselves, the data handled by an application are implemented using C++ classes representing the common VDM data types (sets, maps, sequences etc.). In some cases the formal models for features of VDMTools have been used to generate the production code implementing the features. Furthermore the tests developed for the specification are re-used to test the implementation. This provides a certain degree of confidence that no more errors are introduced during the implementation phase.

4. Industrial Usage

VDM and VDMTools have been applied in a wide variety of application domains. Here we list some of the best documented applications that are in the public domain:

ConForm: An experiment at British Aerospace comparing the conventional development of a trusted gateway with a development using VDM (30; 11).

Dust-Expert: A project carried out by Adelard in the UK for a safety related application determining that the safety is appropriate in the layout of industrial plants (13; 43).

The development of VDMTools: Most components of the VDMTools tool suite are themselves developed using VDM. This development has been made at IFAD in Denmark and CSK in Japan (29).

SIC2000: A project carried out by GAO in Germany for integrating sensor software and hardware in a banknote processing machine (42).

ISEPUMS: In a project from the space systems domain, VDM was used in processing the messages communicated to the SPOT4 satellite (41).

TradeOne: Key components of the TradeOne back-office system developed by CSK systems for the Japanese stock exchange were developed using VDM. Comparative measurements exist for developer productivity and defect density of the VDM-developed components versus the conventionally developed code (18).

FeliCa Networks: This is the development of an operating system for an integrated circuit for cellular telephone applications (28).

A large number of other applications, mainly from safety critical sectors, can not be publicly reported in any detail.

5. Availability and Platforms

VDMTools are available for several different platforms including Windows, Linux, Solaris (for Intel PC) and MacOS. Executables and manuals can be downloaded from http://www.vdmtools.jp/en/ after registration. Free academic licenses can be obtained for bona fide institutions completing a license agreement. In addition, free industrial license agreements can be obtained from CSK Systems.

6. Future Plans

Any tool that is used actively is extended in order to enhance its usability. For VDMTools the most important extensions currently are as follows:

- Because of the importance placed on documentation using formal VDM models, it is planned that a "VDMdoc" feature inspired by JavaDoc (34) will be incorporated as one of the new features.
- Additional standard libraries are desired by existing users and so new extensions in this direction are expected.
- Test automation is an important issue. Here it is expected that VDMTools will be enhanced with new capabilities. Initially the target here is the UniTesK approach (8) for ensuring easy test sequencing automation. However, different test automation possibilities exists and these are currently under investigation.
- Proof support enhancements have been developed for the Overture project (37) and it is envisaged that in the future this will also be incorporated into VDMTools (47).

Significant industrial take-up, for example in Japan, is likely to remain a significant driver for future tools work in VDM. The Overture open-source initiative is more geared towards providing a vehicle for applications and tools research, especially in areas such as proof. Alongside this, new academic courses are being developed that emphasise the transferable skills of abstraction and rigorous reasoning through formal modelling technology (31). Our hope is that, with a strong record of industry application, formalisms such as VDM will continue to contribute to the wider adoption of advanced software development technologies in many application domains.

References

[1] J.-R. Abrial. *The B Book – Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[2] Michael Andersen, René Elmstrøm, Poul Bøgh Lassen, and Peter Gorm Larsen. Making Specifications Executable – Using IPTES Meta-IV. *Microprocessing and Microprogramming*, 35(1-5):521–528, September 1992.

[3] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *Z to B Conference / Nantes*, pages 334–354, 2005.

[4] Bernhard K. Aichernig and Peter Gorm Larsen. A Proof Obligation Generator for VDM-SL. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[5] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[6] Paulo Borba and Silvio Meira. From VDM Specifications to Functional Prototypes. *Journal of Systems Software*, 21:267–278, 1993.

[7] Paulo Borba and Silvio Meira. A System for Translating Executable VDM Specifications into Lazy ML. *Software: Practice and Experience*, 27(3):271–289, 1997.

[8] Igor B. Bourdonov, Alexander S. Kossatchev, Victor V. Kuliamin, and Alexander K. Petrenko. Unitesk test suite architecture. In *FME 2002: Formal Methods – Getting IT Right*, Copenhagen, July 2002. FME, Springer-Verlag.

[9] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–62, April 1995.

[10] Peter Breuer and Jonathan Bowen. Towards correct executable semantics for z. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop*, pages 185–209. Springer-Verlag, 1994. Cambridge.

[11] T.M. Brookes, J.S. Fitzgerald, and P.G. Larsen. Formal and Informal Specifications of a secure System Component: Final Results in a Comparative Study. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 214–227. Springer-Verlag, March 1996.

[12] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[13] Tim Clement, Ian Cottam, Peter Froome, and Claire Jones. The development of a commercial "shrink-wrapped application to safety integrity level 2: the dust-expert story. In *Safecomp'99*, Toulouse, France, September 1999. Springer Verlag. LNCS 1698, ISBN 3-540-66488-2.

[14] Susan Gerhardt Dan Craigen and Ted Ralston. *An International Survey of Industrial Applications of Formal Methods*, volume Volume 2 Case Studies. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, USA, March 1993.

[15] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[16] J. S. Fitzgerald and P. G. Larsen. Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays*, pages 237–254, Volume 4700, September 2007. Springer, Lecture Notes in Computer Science. ISBN 978-3-540-75220-2.

[17] J. S. Fitzgerald, P. G. Larsen, S. Tjell, and M. Verhoef. Validation Support for Real-Time Embedded Systems in VDM++. Technical Report CS-TR-1017, School of Computing Science, Newcastle University, April 2007. Revised version in Proc. 10th IEEE High Assurance Systems Engineering Symposium, November, 2007, Dallas, Texas, IEEE.

[18] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[19] J.S. Fitzgerald and C.B. Jones. Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant. In J.C. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT Series. Springer-Verlag, 1998.

[20] Brigitte Fröhlich. *Towards Executability of Implicit Definitions*. PhD thesis, TU Graz, Institute of Software Technology, September 1998.

[21] Brigitte Fröhlich and Peter Gorm Larsen. Combining VDM-SL Specifications with C++ Code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 179–194. Springer-Verlag, March 1996.

[22] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[23] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

[24] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[25] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.

[26] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[27] Aaron Kans and Clive Hayton. Using ABC To Prototype VDM Specifications. *ACM Sig Plan Notices*, pages 27–37, January 1994.

[28] Taro Kurita, Toyokazu Oota, and Yasumasa Nakatsugawa. Formal specification of an embedded IC for cellular phones. In *Proceedings of Software Symposium 2005*, pages 73–80. Software Engineers Associates of Japan, June 2005. (in Japanese).

[29] Peter Gorm Larsen. Ten Years of Historical Development: "Bootstrapping" VDMTools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.

[30] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[31] Peter Gorm Larsen, John S. Fitzgerald, and Steve Riddle. Learning by Doing: Practical Courses in Lightweight Formal Methods using VDM++. Technical Report CS-TR:992, School of Computing Science, Newcastle University, December 2006.

[32] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*. VDM Europe, Springer-Verlag, March 1991.

[33] Shaoying Liu and Hao Wang. An automated approach to specification animation for validation. *Journal of Systems and Software*, 80:1271–1285, 2007.

[34] Sun Microsystems. JavaDoc homepage. `http://java.sun.com/j2se/javadoc/`, 2007.

[35] Paul Mukherjee. Computer-aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.

[36] Paul Mukherjee, Fabien Bousquet, Jerome Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[37] Overture-Core-Team. Overture Web site. http://www.overturetool.org, 2007.

[38] Overture Group. The VDM Portal. *http://www.vdmportal.org*, 2007.

[39] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.

[40] Nico Plat. The Industrial use of VDM++. In *IEE Colloquium on Industrial Use of Formal Methods*. IEE, May 1997.

[41] Armand Puccetti and Jean Yves Tixadou. Application of VDM-SL to the Development of the SPOT4 Programming Messages Generator. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 127–137, September 1999.

[42] Paul R. Smith and Peter Gorm Larsen. Applications of VDM in Banknote Processing. In John S. Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice: Proc. First VDM Workshop 1999*, September 1999. Available at www.vdmportal.org.

[43] Sunil Vadera, F. Meziane, and M. Huang. Experience with mural in formalising dust-expert. *Information and Software Technology*, 43:231–240, 2001.

[44] S.H. Valentine. Z^{--}, an executable subset of Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 157–187. Springer-Verlag, 1992.

[45] Marcel Verhoef and Peter Gorm Larsen. Interpreting Distributed System Architectures Using VDM++ – A Case Study. In Brian Sauser and Gerrit Muller, editors, *5th Annual Conference on Systems Engineering Research*, March 2007. Available at http://www.stevens.edu/engineering/cser/.

[46] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162. Lecture Notes in Computer Science 4085, 2006.

[47] Sander Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department, August 2007.